Pocket Guide

FORTH

Steven Vickers

Programming Pocket Guides

FORTH

Who the Guide is for and how to use it

This Pocket Guide is designed as a reference book, containing any information about FORTH that you might want to know *quickly*.

The first part is a description of the essence of FORTH, how it compares with other computer languages and what kind of programming you'd use it for. If you've never seen FORTH before but you have enough experience of computer languages to know what computers can do, this part will give you some flavour of the language. It also mentions some of the different dialects you might find.

After that comes a series of brief sections on specific points and, finally, there is a glossary of FORTH words. The glossary contains the fullest details; the preceding sections contain mostly references to the relevant words in the glossary and details on more general points that don't fit easily under any word.

If you already know two or three computer languages, you could probably learn FORTH from this Guide. Less experienced programmers would need a more comprehensive introduction.

Notation

To set them off from English text, FORTH words (this is a technical term that covers symbols as well as ordinary keywords) are printed in **BOLD FACE TYPE**. FORTH words can be found in the **Glossary**.

Zero (0) should be carefully distinguished from capital O.

Outline

The crucial features of FORTH, giving the language its whole character, are the dictionary and the stack.

All your typing is interpreted through the dictionary. It contains all system keywords, variables, arithmetic operators, functions and routines, whether provided in the original system or since by you in your programs.

Because you have so much freedom in what you put in the dictionary, FORTH programming has the flavour of extending the original language into a special-purpose one for your own application. Thus to create a telescope control language (for instance) you first define the utilities DEGREES, AZIMUTH, ELEVATION, OPEN-SHUTTER, SECONDS and CLOSE-SHUTTER, and then the program

```
29 DEGREES AZIMUTH  42 DEGREES ELEVATION
OPEN-SHUTTER 10 SECONDS CLOSE-SHUTTER
```

is self-explanatory. This can be either typed in for immediate execution, or used as a definition of a new word in the *colon* construction:

```
: PHOTO 29 DEGREES AZIMUTH 42 DEGREES ELEVATION
  OPEN-SHUTTER 10 SECONDS CLOSE-SHUTTER ;
```

: and ; mark the beginning and end of the definition of a new word PHOTO.

Although most obvious with control applications, this 'middle-out' approach (design a special-purpose language, then write it in terms of FORTH and your real program in terms of it) works well with all computer programs, and the FORTH dictionary makes it natural.

The other most noticeable feature is the *stack*, a temporary storage area for numbers. Most computer systems have a stack, and in FORTH you have direct control over it. Because you must use the stack directly, FORTH expressions are written in *reverse Polish* notation instead of ordinary infix notation (for example, 3 2 + instead of 3+2). This is a very flexible way of handling numbers. It is often natural and convenient, but sometimes irksome.

A disadvantage with FORTH is that most implementations do not provide arrays, strings, floating point numbers and many other features that even BASIC has. You are expected to add them yourself. Fortunately, there are some FORTHs with these desirable extras already provided.

FORTH always works very interactively: you type in words and numbers and the system responds straight away. This makes debugging very easy, since you can test your new words individually from the keyboard.

Dialects

The main dialects are FORTH-79 (now being replaced by FORTH-83) and fig-FORTH.

FORTH-79 and FORTH-83 are successive official standards specifying a minimal set of required words, some optional sets of words, and their definitions.

The standards are not compatible, notably in division and DO loops. See **/MOD DO ." ' FIND EXPECT NOT PICK ROLL**

fig-FORTH ('fig' stands for FORTH Interest Group) is an implementation of FORTH that is almost entirely written in FORTH, leaving only a few very fundamental words to be written in the machine code of the computer it is to be used on. This means that it is very easy to put fig-FORTH onto most computers — and indeed this has already been done. The fact that fig publish it freely with no copyright restrictions is no small help to its spread.

fig-FORTH conforms generally (although not completely) with FORTH-79, but it also contains all the words it uses in defining itself. These are rather too many to be included in the **Glossary** (but see **fig-FORTH**).

MVP-FORTH (from Mountain View Press Inc.) is essentially fig-FORTH, adapted to conform with FORTH-79 and improved at the lower levels.

polyFORTH is a commercial system produced by FORTH Inc. for minicomputers and includes extra features such as the current date and multitasking. It is well known as the FORTH on which the book *Starting FORTH* by Leo Brodie is based. In general it conforms with FORTH-79; the principal deviations are in the words **'** **,** **."** and **R@**.

MMSFORTH (for the TRS-80) conforms with FORTH-79, but has in addition strings, arrays, a random number generator and floating point arithmetic available.

The Jupiter Ace has its own FORTH in ROM. This is largely FORTH-79, but with rather different I/O (specially designed for cassette tape) and a sophisticated editing system.

The dictionary

The dictionary is the list of the words that have definitions, which define how the words should be executed. A word is a string of any characters other than spaces, and they have the property that if several words occur in one line of input, they are separated by spaces; for example, in the line

```
DEFINITIONS   -69        ZGLYXX   ;
```

there are four words. Two of them (**DEFINITIONS** and **;**) are defined in the standard dictionary, one (**-69**) is a number (or — more precisely — a numeric word, since to the computer the numbers themselves are their binary representations) and the other word (**ZGLYXX**) is neither defined nor a number.

Defining words are used to define new words.

There is a limit on the lengths of words that can appear in the dictionary, 31 characters in the standards. Some older FORTHs (for example, polyFORTH), when defining a word in the dictionary, don't store all its characters but just the first three together with a note of the total length. Thus **CONTEXT** and **CONVERT** would both be stored as '7 letter word beginning with **CON**' and could not be distinguished. A few systems treat capitals and lower-case letters as interchangeable.

When a word is defined more than once in the dictionary, the newest definition overrides the earlier ones — but only for subsequent uses. Words defined before the newest definition will still use the older ones. If a word could also be interpreted as a number (for example, if **-69** had a dictionary definition), then the dictionary definition overrides the numeric interpretation.

Each defined word has associated with it two numbers, its *compilation address* and its *parameter field address*. These indicate where the definition is stored in memory and can be used as numeric codes for the word. Beware that different FORTHs vary on which of the two addresses they use in a given context. See Compilation, Extending FORTH and fig-FORTH, and **' ['] FIND >BODY EXECUTE**.

The dictionary can be subdivided into **vocabularies**; when it is, all dictionary searches follow the rules set out in that chapter.

See also **FORGET**.

The data stack

(There is also a return stack; but 'stack' unqualified is always the data stack.)

Numbers that are required only temporarily (more permanent ones would be assigned to variables) are kept in a *stack*, a list of numbers in which the last to be put in the stack — the *top* of the stack — is the most readily available for anything that needs one of the numbers. When a number is typed at the keyboard for immediate execution, it goes straight on top of the stack.

When a defined word is executed, it can take some numbers, its *arguments*, off the top of the stack, do its calculations, and can put some new numbers, its *results*, back on the stack. The arguments are thrown away and no longer stored on the stack.

Since the stack entries are not differentiated except by their position, it is important to keep track of them — otherwise it is easy to leave entries on or take them off by mistake. A standard notation is to list the entries from the bottom to the top. For example, in

2 3 + .

the stack goes through the following states:

Word	*Stack* *(bottom)*	*(top)*
2	2	
3	2,	3
+	5	(+ has two arguments and one result)
.		(. has one argument and no results)

Any numbers on the stack before 2 would still be there, untouched, after .

See **?DUP DUP 2DUP DROP 2DROP OVER 2OVER ROT 2ROT SWAP 2SWAP PICK ROLL DEPTH .S**

The return stack can often be useful in stack manipulations.

Arithmetic

- Because you use the stack directly, expressions are written in *reverse Polish* notation, as on Hewlett Packard pocket calculators.
- All numbers are *integers*.
- The arithmetic is best understood in terms of the *binary representation* of the numbers; but you can still use FORTH without this depth of knowledge.
- There is very little error checking in arithmetic.

Reverse Polish notation

Reverse Polish notation writes operators *after* their arguments, for example 2 2 + for 2+2, or 2 3 + 4 5 + * for (2+3)*(4+5), and is used in FORTH because an operator expects to find all its arguments already on the stack. It never needs brackets — in fact, brackets are used for comments in FORTH.
To convert a reverse Polish expression to infix (ordinary) go through the expression one word at a time, writing down exactly what will be on the stack and putting brackets in where necessary. For example:

Word	*Stack*
2	2
3	2, 3
+	2+3
4	2+3, 4
5	2+3, 4, 5
+	2+3, 4+5
*	(2+3)*(4+5)

To convert an infix expression to reverse Polish

- Ask what is the *last* operation to be performed (* in the example).
- Ask what are the numbers required on the stack for this last operation — (2+3) and (4+5).
- If as in this case they are expressions rather than pure numbers, then convert them separately into reverse Polish by considering their last operation and so on.

- Write down the two numbers or converted expressions one after the other, and then the last operation:

- Convert the reverse Polish expression back to infix as a check.

Integer arithmetic

Standard FORTH only uses *integers*; that is, whole numbers (positive or negative).

There is an important difference between FORTH-83 and the older FORTHs in the way that they round the result of an inexact division when it is negative. See **/MOD**.

Binary representation and how to ignore it

For the sophisticated programmer:

- Numbers are stored in *binary* using 16 bits.
- Hence *bitwise Boolean* operations can treat them as *bit patterns*. See **AND OR XOR NOT 2/ 2***
- Negative numbers are stored in *twos complement* form.

These concepts are dealt with in *Microprocessor Fundamentals* by Halsall and Lister, published by Pitman.

If you're not familiar with them and don't want to get bogged down in their technicalities, read the rest of this section.

Signed and unsigned numbers The computer can handle numbers between −32768 and 65535, but it can confuse the negative numbers (−32768 to −1) with the large positive ones (32768 to 65535), for instance:

65535 . displays '–1'.

Numbers between −32768 and 32767 are called *signed*.

Numbers between 0 and 65535 are called *unsigned*.

Because of this confusion, for some operations you need to use different words in the two cases, like the words **.** and **U.** (U for Unsigned) that display signed and unsigned numbers. This is why **65535 .** gave the wrong answer — it used **.** with an unsigned number. Use instead **65535 U.**

Naive rules of thumb to avoid trouble

- For numbers that are both signed and unsigned (0 to 32767), don't worry.
- When numbers are involved that are either signed or unsigned but not both (that is, between −32768 and −1 or between 32768 and 65535), check your operations in the Glossary to see what kind of arguments they require.
- For numbers that are neither signed nor unsigned (because they are positive and too big or negative and too small) the computer will give you nonsense results without warning you. Some examples that produce this trouble — *overflow* — are

−30000 20000 − (answer −50000 is too small to be signed)
300 400 * (answer 120000 too big to be unsigned)

Double length arithmetic

The numbers seen so far (signed or unsigned), taking up one stack entry (16 bits) each, are called *single length* numbers. You can also use two stack entries together (32 bits) to hold much larger *double length* numbers.

To convert a number from single length to double length:

- For an unsigned number, put a 0 on top of it on the stack.
- For a negative number, put −1 on top of it on the stack.

Examples

Single length number (one stack entry)		*Double length equivalent (two stack entries)*
60000	(unsigned)	60000, 0
150	(signed or unsigned)	150, 0
0		0, 0
−1	(signed)	−1, −1
−30000		−30000, −1

Double length arithmetic deals with unsigned integers between 0 and 4294967296 or signed integers between −2147483648 and 2147483647. These can be written in 32 bits and stored in two halves so that the value represented is

less significant half + 65536 * more significant half

On the stack, the more significant half is on top. Note the double length stack manipulations **2DUP**, **2DROP** etc.

Arithmetic words

See **+ − 1+ 1− 2+ 2−**
*** / */ MOD /MOD */MOD 2* 2/**
= < > U< 0< 0= 0> NOT
ABS MAX MIN NEGATE MINUS

Double length words are usually the same but starting with D or 2 (e.g. **D+**, **2DUP**). Some words act partly on single length and partly on double length numbers. These usually start with M ('mixed'), UM, or sometimes just U.

Flags

A word, like **<**, that performs some kind of test ('Is the number second from top on the stack less than the number on top?') gives as its result a *flag*, one of only two values: *true* ('Yes, it is less') or *false* ('No, it's not'). *True* and *false* are just ordinary numbers, although precisely what numbers depends on the FORTH system.

False is always 0.
True is −1 in FORTH-83, 1 in all older FORTHs.

Flags can be combined using *Boolean* (or *logical*) operations:

AND OR XOR = 0= NOT

When one of these words is described in the Glossary as *bitwise* Boolean, that means it can also be applied to bit patterns.

Words that take a flag off the stack (for example, **IF**, **UNTIL**) are prepared to accept any non-zero number as *true* (only 0 is *false*). This does not apply to the bitwise Boolean words.

Programs

In FORTH, both programs and subroutines are covered by the concept of the *colon definition*: a stretch of defined words and numbers that are compiled to make them the definition of a new word (the program or subroutine). See : .

Program structure

Within a colon definition (but not in normal executed text) you can use various control structures similar to those in other languages:

IF ... **ELSE** ... **THEN** for branching
BEGIN ... **UNTIL** or **BEGIN** ... **WHILE** ... **REPEAT** for repetitions
DO ... **LOOP** for counted repetitions (like FOR ... NEXT in BASIC)

You cannot spread such a structure across several colon definitions; and although you can have one structure *nested* completely inside another and you can do this nesting to any depth you like, you mustn't let two structures cross. For example:

Allowed **IF** ... **BEGIN** ... **UNTIL** ... **ELSE** ... **THEN** (in which **BEGIN** ... **UNTIL** nests between **IF** and **ELSE**)

Not allowed **IF** ... **BEGIN** ... **ELSE** ... **UNTIL** ... **THEN** (**BEGIN** ... **UNTIL** crosses from one half of **IF** ... **ELSE** ... **THEN** to the other)
nor **BEGIN** ... **IF** ... **UNTIL** ... **ELSE** ... **THEN**
See also compiling words.

Other program control words **EXIT QUIT ABORT ABORT"**

Comments

Comments, to be ignored by the computer, are enclosed between round brackets. See (.

Memory manipulation

The FORTH system assumes a 64K byte memory address space which need not, though it often does, coincide with the address space of the computer. This means that a single length unsigned integer can be used as the address of a byte of memory. Two consecutive bytes of memory, addressed by the lower of their two addresses, are called a *cell*. When a single length (2-byte) integer is stored in a cell, the order of the two bytes varies from computer to computer (see **><**). When a double length (4-byte) number is stored in two consecutive cells, the more significant half is stored in the first cell.

See **C@ C! @ ! 2@ 2! ? CMOVE CMOVE> FILL DUMP MOVE <CMOVE ERASE +!**

Variables

Normal FORTH programs use variables less than other languages, because there is the stack instead. They are kept in the dictionary.

See **VARIABLE**, and also **CONSTANT 2VARIABLE 2CONSTANT USER** .

Input/output

The text interpreter

The *input stream* (either from the keyboard or from a mass storage device; see **BLK** and **>IN**) is analysed word by word by the *text interpreter*, either in execute mode, for immediate execution, or (after words such as **:**) in compile mode, in which it builds up a colon definition.

A word is first looked up in the dictionary. If it is found, and the interpreter is in execute mode, the word is executed; if in compile mode, the word's compilation address is enclosed in the dictionary (see compiling word). However, some words, typically compiling words, are specified to be *immediate*. This means that they are executed immediately even if the interpreter is in compile mode. (See **IMMEDIATE [COMPILE]**.)

In some FORTHs there are state dependent immediate words whose action depends on the interpreter's mode; but FORTH-83 has eliminated these. (See ' ," , **STATE** .)

If a word is not defined, then the interpreter tries to interpret it as a number according to the system number base. A number should be single length, in the range −32768 to 65535 (it is not defined what happens if the word represents a number outside this range), and may start with a minus sign but not a plus sign. In execute mode the number is put straight on the stack, while in compile mode it is enclosed in the dictionary after a compilation address that will stack it when the new colon definition is executed. (See compiling word.)

In some FORTHs (not the standards) a word can be interpreted as a double length number if it contains certain characters (which do not otherwise affect the value of the number), for example '123'. for double length 123 or '80.000' for correctly stored 80000. In polyFORTH, the five characters ',', '.', '/', '−' (except initially) and ':' all serve this purpose.

If a word is neither defined nor a number, the interpreter will give an error message.

At the end of a line (or of a mass storage buffer), the interpreter will display 'OK' if it is in execute mode and all went well. Note that a few older FORTHs, if they reach the end of the line in compile mode, automatically revert to execute mode. Then, if you're still in a colon definition, you must start the next line with].

Using the input stream

The text interpreter takes its words from the input stream, but there are also a number of FORTH words that can do this independently of the interpreter. For instance, defining words like **VARIABLE** take another word, the name of the variable, from the stream and enclose it as a word followed by a definition: for example, **VARIABLE CEPHEID**.

Since **VARIABLE** is in effect taking the word **CEPHEID** as an argument, this shows how non-numeric arguments, like words, can be passed through the input stream instead of the stack. Such word arguments have some important differences from normal numeric arguments.

- They rely on the word that uses them to take them out of the input stream, so the word (for example **VARIABLE**) precedes its word argument (**CEPHEID**), unlike the normal reverse Polish convention. That is why you don't say **CEPHEID VARIABLE**.
- Word arguments cannot be compiled into colon definitions. You cannot say

: INITIALIZE VARIABLE CEPHEID ;

The fundamental words for handling input stream words are **WORD** and **CREATE**.

Once a word is defined, it can be handled by being converted to its compilation address or parameter field address, using **'** , **[']** or **FIND**. After the conversion it can be manipulated on the stack like any other number.

See also **NUMBER**, **INTERPRET** and **CONVERT**.

For collecting typed input, see **QUERY**, **EXPECT**, **SPAN**, **KEY**, **TIB**, **#TIB** and **>IN**.

Mass storage

Mass storage in FORTH usually means disks, where you keep the text for your FORTH programs as well as any other data you need.

The disk is divided up into numbered *blocks* of 1024 bytes (sometimes called *screens*). When you need the data from a block, it is read into a *block buffer* in memory (see **BLOCK**, and also **BUFFER**) where you are free to read or modify it.

Because changes are not written to disk straight away, it is important to make sure that the disk is up to date before you ever remove it: do this with **FLUSH**. Also use **FLUSH** at the end of any editing session, and **SAVE-BUFFERS** periodically during it, to keep the disk up to date in case of a power failure or other system crash.

A program on disk is just stored typing, uncompiled, and **LOAD** makes the text interpreter read it as though you were typing it all in again yourself. (See also **BLK** and **>IN**.) Either the text interpreter runs through to the end of the block and returns to the keyboard, or at some stage it comes across **--->**.

See also **THRU EMPTY-BUFFERS LIST SCR INDEX EDITOR**.

Number bases

Number bases are explained in many books on computing.

All numeric input and output uses the *system number base*, the value of the variable **BASE**. Normally this is 10, to give decimal notation, but it can be changed to any base between 2 and 71. The characters used as digits in bases larger than ten are first the capital letters and then the rest of the ASCII characters from [to }.

Examples:

DECIMAL 243 HEX .

displays 'F3' (hexadecimal notation for 243).

2 BASE ! −1001010 DECIMAL .

displays '−138'.

Note that once a number is compiled into a colon definition, it keeps its value regardless of what happens to the system number base. For example:

: DUODECIMAL 12 BASE ! ;

If you now do **HEX** (base sixteen), the number 12 compiled into the definition of **DUODECIMAL** remains twelve: it doesn't change to eighteen.

See **BASE DECIMAL HEX OCTAL H. O.**

Terminal output

To output characters and strings **EMIT TYPE SPACE SPACES ."**
To output numbers **. .R H. O.** (for signed numbers)
U. U.R (for unsigned numbers)
D. D.R (for double length numbers)

Numbers are printed according to the system number base, with a minus sign in front if negative. A space is displayed afterwards (except with **.R**, **U.R** and **D.R**).

See also formatted output.

Terminal control words **CR PAGE**

Formatted output

An unsigned double length number can be displayed in a controlled way, perhaps interspersed with other characters, using *formatted output* between <# and #>. You consider from *right to left* what characters you want to be finally displayed and specify this from *left to right* between <# and #>, using # for the next digit from the double length number and **HOLD** for a specified ASCII character. **#S** gives all the remaining digits. The digits and characters are stored as a text string somewhere in memory, and #> leaves the address and length of the string.

A typical example is currency. If there are 100 widow's mites to the piece of eight (symbol 'P'), then you would record an amount of money in widow's mites but print it with a decimal point so that it appears in pieces of eight, for example:

P2942.76

Work from right to left. You want two digits, then a decimal point, then the remaining digits, then P, and finally to display it all. This is done by

```
<#  #  #      ( two digits for widow's mites)
  46 HOLD     ( decimal point)
  #S          ( remaining digits for pieces of eight)
  80 HOLD     ( P)
#>  TYPE
```

The number being printed is unsigned double length (FORTH-83 requires it to be both signed and unsigned). #, **#S** and #> expect it to occupy the top two stack entries, and for each digit taken out of it, it is divided by the system number base. After **#S** it will have gone down to 0.

For signed numbers, see **SIGN**.

For lengthening of single length numbers, see double length numbers.

See also **ASCII**.

Vocabularies

It sometimes helps to be able to split the dictionary up into parts (*vocabularies*) so that the same word can have different definitions in different vocabularies. Typical examples are the editor and assembler vocabularies (not standard, but present in most implementations). When you want to use the editor, you switch over to the editor vocabulary with **EDITOR**. This puts the editor words at your disposal (it puts the text interpreter into the editor *search context*), and although some of them (for example, **I**) may be the same as normal FORTH words, they will use the editor definitions.

Each vocabulary has a name, a FORTH word that makes its own words available (like **EDITOR**): it makes it the (search) context vocabulary. The standard vocabulary is called **FORTH**.

Any dictionary search for a word will start off in the context vocabulary; but if the word is not there the search may try a different vocabulary. Which further vocabularies are tried, and in what order, varies from FORTH to FORTH. FORTH-83 makes no requirements apart from recommending that the search should finish up with the **FORTH** vocabulary. FORTH-79 recommends that the search should try first the context vocabulary, then **FORTH** (this is described by saying that all vocabularies *chain* to **FORTH**).

In fig-FORTH a new vocabulary will chain to the vocabulary in which it was defined. For instance, the vocabulary word **EDITOR** is itself in the **FORTH** vocabulary and so chains to **FORTH**. Any unsuccessful search through **EDITOR** will follow up with **FORTH**.

The vocabulary that takes in new definitions is called the *current* vocabulary. Note that : changes the search context to that of the current vocabulary.

In fig-FORTH, vocabulary words are immediate. The standards are not explicit on this, although in FORTH-79 **FORTH** is immediate.

See **VOCABULARY DEFINITIONS CURRENT CONTEXT FORTH EDITOR ASSEMBLER**.

Extending FORTH

Defining words

A word definition falls into two parts, the *header* and the *parameter field*. These were added to the dictionary, or *enclosed* in it, by the *defining word* that set up the definition.

The parameter field is an area of memory containing the information special to the definition: the value of a constant or variable, the list of compiled words in a colon definition (see compiling words), etc. The address of its first byte is the *parameter field address* (PFA).

The *header* contains information saying what to do with the parameter field: get the value and put it on the stack (for a constant), put the address on the stack (for a variable) or set the address interpreter on it (for a colon definition) etc. The header also contains the word itself and other information needed by the system. The compilation address is usually the memory address of part of the header; but the standards do not specify this in any way and according to them it is just a code number associated with the word definition. (But see fig-FORTH.)

The defining words in the glossary are **VARIABLE 2VARIABLE CONSTANT 2CONSTANT USER : CREATE VOCABULARY**.

CREATE (or **<BUILDS**) is used to make words with parameter fields to your own design. See also **DOES<** , **C, ALLOT HERE**

Compiling words

To set up a colon definition, the text interpreter goes into compile mode and simply encloses in the dictionary the compilation addresses of the defined words that it comes across: so the colon definition is a list of these *compiled words*. For numbers, it encloses a compound structure comprising the compilation address of a *literal handler* (a routine to stack the number) followed by a *literal operand*, the number itself. This combination is still called a compiled word.

When the colon definition is executed, an *address interpreter* takes the compiled words one by one and executes the corresponding definitions. The literal handler stacks the number and makes the address interpreter skip over it to the next compiled word.

Note that the literal handler may be *anonymous*, a definition without a word, like 'the back of the knee' in English; but it can still have a compilation address and be referred to by that.

Words like **IF** that are immediate and modify the enclosing process are called *compiling words*. The immediate, *compile-time* action of **IF** is to enclose a compiled word consisting of a compilation address (that of **?BRANCH** in FORTH-83) followed by some space for a literal operand that **ELSE** or **THEN** later fills in to show its own position. The *run-time* action, the execution of this compiled word, is to force the address interpreter either to skip over the literal operand or to use it to jump to **ELSE** or **THEN**.

Compiling words in the glossary are program structure words, **LITERAL**, **."**, **[']** (or **'**), **ABORT"** and **COMPILE**.

In the Glossary, the definitions whose compilation addresses require a literal operand are **BRANCH**, **?BRANCH** and the run-time actions of some of the compiling words.

There is no standard way of defining more of these words that, when compiled, require a literal operand, although it can usually be done by adjusting the return stack.

Note that many compiling words use the data stack at compile time to pass information to other compiling words (*cf.* **>MARK** and **>RESOLVE**). This includes syntax checking codes to ensure that the compiling words match up properly.

Compiling words often need to use **[COMPILE]** (before immediate words) and **COMPILE** (before others). For example, to define a compiling word **UNLESS** to do **0= IF** , **: UNLESS COMPILE 0= [COMPILE] IF ; IMMEDIATE**

See **[ ]**.

The return stack

There is a second stack, called the *return stack*. Suppose the address interpreter is processing the compiled words in a colon definition. It remembers for itself a number called the *instruction pointer*, an address showing whereabouts in the colon definition it is looking. This more often than not is the address of the compiled word currently being processed plus two (to take it to either the next compiled word or the literal operands) but the standards don't specify this. If the compiled word refers to another colon definition, the instruction pointer is temporarily stored on the return stack (as a *return address*) while the address interpreter considers instead the instruction pointer for this other colon definition; when this is finished, the old pointer is retrieved from the return stack so that the interpreter can carry on where it left off.

The standards don't specify this mechanism precisely, so its main use is as a temporary store for items that would be in the way if left on the data stack.

This requires care because many FORTHs store there not only return addresses but also values and limits for **DO** loops. This means that the return stack must be in the same state at **LOOP**, **+LOOP**, **LEAVE**, **I**, **I'**, **J** or **K** as it was at the corresponding **DO**, and in the same state at **;** or **EXIT** as it was at the corresponding **:**. You must thus be careful to balance **>R**s and **R>**s within these structures, and you can't use **EXIT** inside a **DO** loop.

See **>R** **R>** **R@**, and **SIGN** for an example.

Recursion is when a word uses itself, either directly or indirectly, through another word. Because FORTH uses stacks, recursion is always possible, but awkward because a word can't be used until it is fully defined. You get round this by initially defining an *execution vector* for the word: a variable that you later fill in with the compilation address of the proper definition. But even before that is done, you can use the word in other colon definitions by the trick

name of execution vector **@** **EXECUTE**

See **RECURSE**.

Strings

Strings are not treated in any coherent fashion, and in most FORTHs only rudimentary facilities are provided. There are two main methods of referring to strings.

Counted strings (up to 255 characters) are stored in memory immediately preceded by a single byte containing the number of characters. To refer to such a string, you use the address of its count byte.

Text strings are stored in memory without a count byte and are referred to by two numbers: the address of the first character (this number is underneath on the stack) and the number of characters (on top).

See **COUNT –TRAILING TYPE ." .(WORD PAD ASCII –TEXT CONVERT FIND NUMBER TEXT**.

It is not hard to write much more useful string handling words.

Arrays

Arrays are not provided in standard FORTH, but there is no technical problem in using **CREATE** to define words with large **ALLOT**ted parameter fields as the storage space for arrays. See also **DOES>**.

Assembly language

Most FORTHs provide an assembler, so that words can be defined in machine code if necessary. Although much obviously depends on the processor in use, there are some general points.

- The assembly language is usually modified to fit in with FORTH: the mnemonics and operands become FORTH words and are generally written with the mnemonic after the operand, giving it all a backward appearance (like reverse Polish).
- The assembler words will be in an **ASSEMBLER** vocabulary.
- Words will be defined in machine code using **CODE**, **;CODE** and **END-CODE** (analogous to **:**, **DOES>** and **;**).

fig-FORTH

While not absolutely standard, the underlying structure of fig-FORTH is fairly typical.

A header has three fields (and the parameter field makes a fourth).

Name field — a counted string for the word itself (up to 31 characters). The count byte may be modified to show other properties of the word: before the word is fully defined, there is 32 added on; for an immediate word 64 is added on; and for all words 128 is added on.

Link field — two bytes containing the address of the name field of the previous word defined in the word's vocabulary. Dictionary searches follow these links down.

Code field — two bytes containing the address of some machine code that will be called when the word is executed. For *primitive* words defined in machine code (for example, **DUP**, **+**), the machine code will occupy the parameter field. For other words the machine code will be somewhere else, but it will know where the parameter field is so that it can use the information there.

The addresses of the four fields are usually known as the NFA, LFA, CFA (the compilation address) and the PFA.

Errors

FORTH systems do very little error checking, so as to keep execution fast: so no checking on arithmetic (for overflow etc.), and the stack is checked for underflow (an attempt to take a number off an empty stack) or running out of space (stack too full) only between words in the input stream.

If an error happens during the compilation of a colon definition (for example, through wrongly matched compiling words), some systems leave the dictionary in a fragile state, so check that yours behaves sensibly. You may have to use a word **SMUDGE** before you can **FORGET** the incomplete definition, and the context vocabulary may have reverted to **FORTH**.

Glossary

This mentions all the FORTH words mentioned in the FORTH-79 and FORTH-83 standards except for some uncontrolled reference words, as well as most of the words in *Starting FORTH* by Leo Brodie and the most useful words in fig-FORTH. The alphabetical ordering is determined by the order of the ASCII character set.

Notation

For the dialects in which the word is available

All used in all FORTHs, required in both standards
83 FORTH-83
79 FORTH-79, with word sets denoted as follows

R	required word	
DN	double number word	optional
A	assembler word	optional
S	system word (FORTH-83 only)	optional
CR	controlled reference word	optional
UR	uncontrolled reference word	optional

Reference words are not official, but controlled reference words are not supposed to be used except with the meaning in the standard.

poly polyFORTH (Brodie)
fig fig-FORTH

Usage

C only for use in compile mode — normally a compiling word
I immediate
U a user variable (see **USER**)

The effect on the stack is shown by listing the arguments, which the word removes, then '——', then the results, which are left afterwards. For example, for **/**,

n1, n2 —— *n1/n2*

Note that the compile-time stack action for compiling words is not usually shown. Most of these put codes on the stack in an implementation dependent way so that they can check that they are used in correct combinations. **:** is similar.

Abbreviations for stack entries are

flag	a flag
b	bit pattern
n	signed single length number
u	unsigned single length number
w	any stack entry
d	any double length number (two stack entries)
sd	signed double length number (two stack entries)
ud	unsigned double length number (two stack entries)
addr	unsigned single length number used as an address
char	character (7-bit ASCII code)
(addr)	the contents of the memory cell at addr
:=	becomes

Input text Under 'used in the form', lower-case letters refer to text taken from the input stream. For example, **VARIABLE** is used in the form

VARIABLE name

This doesn't mean that **VARIABLE** must also be in the input stream; it could be executed from another word.

Standard pronunciation is often given in double quotes, for example:

>IN "to in"

The words

! *w, addr* —— All "store"
Stores *w* in memory at *addr*, typically used with variables.
This is a dangerous word, albeit very necessary, because it is easy to have the wrong address on the stack and to overwrite something by mistake. Note the order of its arguments. The standard analogy is with making up a parcel, then writing the address on top.

*ud* —— *ud*/(**BASE**) All "sharp"
Used in formatted output to produce one digit from ud.

#> *ud* —— *addr, length* All "sharp greater"
Used in formatted output to produce the final text string.

#S *ud* —— 0, 0 All. "sharp S"
Used in formatted output to apply # repeatedly until the double length number on the stack is 0. If *ud* = 0, then # is applied once to produce a single 0 digit.

#TIB —— *addr* 83R. "sharp T I B" U
A user variable whose value is the total number of characters in the text input buffer. **#TIB** does not exist in other FORTHs; in them the end of the buffer is marked by at least one null character (ASCII 0).

' —— *addr* All. "tick"
Used in the form ' word
The following word is not executed, but its address is left on the stack. For example:

' DUP U.

prints the address of **DUP**.
Although very important, ' is highly variable in its behaviour. The result can be either the compilation address of the word (83) or its parameter field address (79, poly, fig). If the word is undefined, an error results.
In FORTH-79 and fig-FORTH, ' is a state dependent immediate word. In compile mode it behaves like **[']** (but finding the parameter field address). See also **FIND >BODY**.

'S —— *addr* poly. "tick S"
The result is the address of the top of the stack, not counting the address itself.

(—— All. "paren" I
Used in the form (comment text).
(Note the space necessary after the word (.)
The comment is ignored up to the terminating). It should all be contained in a single bufferful of the input stream, and must not contain a).

***** *w1, w2* —— *w1*w2* All. "times"
Multiplication of signed or unsigned arguments.

***/** *w1, w2, n3* —— *w1*w2/n3* All. "times divide"

w1 and *w2* can be signed or unsigned.

The intermediate product, *w1*w2*, is a signed double length number to maintain accuracy, so ***/** is better than ***** and **/** separately. See **/MOD** for differences between FORTH-83 and the others.

***/** is useful for multiplying numbers by fractions. For instance, π is approximately 3.1416, or 31416/10000, so to find — say — 4 π, do

4 31416 1000 */.

See **2CONSTANT**.

***/MOD** *w1, w2, n3* —— r, q All. "times divide mod"

q and r are the quotient and remainder for the division (*w1*w2*)/*n3*, with *w1*w2* held as a signed double length number as in ***/**. They are calculated according to the rules in **/MOD**.

+ *w1, w2* —— *w1+w2* All.

w1 and *w2* can be signed or unsigned.

+! *w, addr* —— All. "plus store"

Adds *w* to the memory cell at *addr*.

+LOOP (run-time) n —— All. CI

See **DO**.

, *w* —— All.

Encloses *w* as two bytes in the dictionary. See **CREATE**.

– *w1, w2* —— *w1–w2* All.

w1 and *w2* can be signed or unsigned.

––> —— 79CR, 83CR, fig. "next block" I

Used on mass storage blocks. It tells the text interpreter to continue immediately with the next block on disk.

–DUP *w* —— *w, w* fig.
or 0 —— 0
The same as **?DUP**.

–FIND —— *PFA, count byte, true* fig.
or —— *false*
Used in the form — **FIND** word.

If the word is defined, stacks its PFA, the first byte of its name field and *true*; otherwise just stacks *false*.

–TEXT *addr1, length, addr2* —— *n*
79UR, 83UR, poly. "dash text"
The addresses are of two text strings, both of the same *length*. **–TEXT** compares the strings, and *n* is the difference between the first non-matching characters (or 0 if the strings are equal). Thus as *n* is positive or negative, the first string comes after or before the second in alphabetical order.

–TRAILING *addr, length1* —— *addr, length2*
All. "dash trailing"
Adjusts the length of its text string argument so as to ignore any spaces at the end.

. *n* —— All. "dot"
Displays *n* at the terminal, followed by a space. See Terminal Output.

." (run-time) —— All. "dot quote" IC
Used in compile mode in the form

." string"

(Note the space, not part of the string, after the word **."** . The string cannot contain a " character.)

." compiles a string into a colon definition so that at run-time the string is displayed at the terminal. For example:

: **GREET CR ."** Well, blow me down." ;

At compile time, the string must be all in a single bufferful of the input stream.

Note: Except in FORTH-83, **."** can also be used in execute mode, with the effect of **.(** .

.(—— 83R. "dot paren" I
Used in the form .(string)
(Note the space, not part of the string, after the word .(. The string cannot contain a) character.)

Displays the string at the terminal. It is most useful in mass storage blocks, to display messages during their interpretation, for example:

CR .(Starting to compile block 36 now.)

In older FORTHs, use ." instead.

.R *n1, n2* —— 83CR, 79UR, fig, poly.
Displays *n1* at the terminal, with no following space, and enough spaces in front to make up a total field of *n2* characters with *n1* at the right. For example:

−109 7 .R

displays 3 spaces, then the 4 characters '−109'. See Terminal Output.

.S —— None.
Displays the entire stack without changing it.

/ *n1, n2* —— *n1/n2* All. "divide"
Signed division. See **/MOD** for differences between FORTH-83 and the others.

/LOOP (run-time) u ——
83UR, 79UR, poly. "up loop" IC
Like **+LOOP**, but with unsigned arithmetic. See **DO**.

/MOD *n1, n2* —— *r, q* All. "divide mod"
q and r are the quotient and remainder for the division *n1/n2*.

There is an important difference between FORTH-83 and the older FORTHs in the way they divide negative numbers, which applies also to **/**, ***/** and ***/MOD**.

In FORTH-83, division is *floored*, that is the quotient, if not an exact integer, is rounded *down* to an integer (so in dividing −84 by 10 the true answer, −8.4, is rounded down to −9). This implies that the remainder has the same sign as the divisor *n2*.

In older FORTHs, division is *truncated*, that is the quotient is rounded towards zero (so −8.4 is rounded to −8). The remainder then has the same sign as the dividend *n1*.

Note that in neither case is the quotient necessarily rounded to the *nearest* integer. In all cases,

$n1 = n2 * q + r$ and $|r| < |n2|$

Examples:

		FORTH-83		*Older FORTHs*	
n1	n2	q	r	q	r
84	10	8	4	8	4
84	−10	−9	−6	−8	4
−84	10	−9	6	−8	−4
−84	−10	8	−4	8	−4

0< n —— *flag* All. "zero less"
Yields true if n is negative (like **0 <** with a space).

0= n —— *flag* All. "zero equals"
Yields true if n is zero (like **0 =** with a space).

0> n —— *flag* All. "zero greater"
Yields true if n is positive (like **0 >** with a space).

1+ *w* —— *w*+1 **All.**

1− *w* —— *w*−1 83R, 79R, **poly.**

2 or D is often used at the start of a word with a double length operation.

2! *d, addr* ——
83DN, 79DN, fig, poly "two **store**"
or *w1, w2, addr* ——
Double length **!** . (*addr*) := *w2*, (addr+2) := *w1*

2* *w* —— *w*2* 83 CR, 79UR, poly
A bit pattern *w* is shifted left one bit, bringing in a 0 on the right.

2+ *w* —— *w*+2 All.

2– *w* —— *w*–2 83R, 79R, poly

2/ *n* —— *n*/2 83R, 79UR, poly.
Note that this division is always floored (see **/MOD**). A bit pattern *n* is shifted right one bit, the new leftmost bit being the same as the old one.

2@ *addr* —— *(add + 2), (addr)*
83DN, 79DN, fig, poly "two fetch"
Double length version of **@**.

2CONSTANT *d* —— 83DN, 79DN, poly
or *w1, w2* ——
Double length version of **CONSTANT**, a defining word used in the form

d **2CONSTANT** name

When 'name' is executed, it leaves *d* on the stack.
Note that **2CONSTANT** is usable not only for double length numbers, but also for pairs of single length numbers such as fractions for use with ***/**. For example:

31416 10000 2CONSTANT PI
(actually, **355 113 2CONSTANT PI** is more accurate)
4 PI */ .

2DROP *d* —— 83DN, 79DN, poly
w1, w2 ——

2DUP *d* —— *d, d* 83DN, 79DN, fig, poly
or *w1, w2* —— *w1, w2, w1, w2*

2OVER *d1, d2* —— *d1, d2, d1*
83DN, 79DN, poly
or *w1, w2, w3, w4* —— *w1, w2, w3, w4, w1, w2*

2ROT *d1, d2, d3* —— *d2, d3, d1* 83DN, 79DN
or *w1, w2, w3, w4, w5, w6* —— *w3, w4, w5, w6, w1, w2*

2SWAP *d1, d2* —— *d2, d1* 83DN, 79DN, poly
or *w1, w2, w3, w4* —— *w3, w4, w1, w2*

2VARIABLE —— 83DN, 79DN, poly
Double length **VARIABLE**, a defining word used in the form

2VARIABLE name

When executed, 'name' leaves on the stack the address of its 4-byte parameter, to be used by **2@** or **2!**. See **2CONSTANT**.

79-STANDARD —— 79R.
Like **FORTH-83**, but for FORTH-79.

: All. "colon"
A defining word, used in the form

: name text for compilation ;

For example:

```
: SUM ( ——  ) ( Displays 2+2)
  2 2 + .
;
```

The layout here is not important, except that **SUM** must follow **:** on the same line.

After taking the word to be defined (**SUM** in this case), **:** puts the text interpreter into compile mode, so that **2 2 + .** is compiled into the dictionary to form a colon definition for **SUM**. (The comments in round brackets are ignored.) Finally, **;**, an immediate word, completes the colon definition and restores the text interpreter to execute mode.

When the new word **SUM** is executed, it will add 2 and 2 and display the result.

: is one of the most important words in FORTH, because the colon definition plays the role of program or subroutine in other languages: thus **SUM** may be an entire program, or it may be just a subroutine to be used in further words. In either case, it can be run or tested from the keyboard just by entering **SUM**.

Note the comments, which specify *precisely* what **SUM** does to the stack and then give a more general description. These are crucial for easy debugging, since without them it is easy to define words to do one thing to the stack but use them as though they do something different.

Within a colon definition there is a wide variety of program structures and compiling words available. See also **[ ] LITERAL**

Note that in most FORTHs, **:** makes the search context start with the current vocabulary.

While the word is still being defined, it is smudged; that is, it is not recognized as a defined word (unless it has an older definition, in which case that is still used).

; All. "semicolon" IC

Finishes the compilation of a colon definition. See **:**.

;CODE 83A, 79A, fig. IC

Used in the form

: namex . . . **CREATE** . . . **;CODE** assembly code **END-CODE**

;CODE is analogous to **DOES>**, so that 'namex' is now a new defining word, used in the form

. . . namex name

However, the action of 'name' is defined not by a FORTH routine, as after **DOES>**, but by the assembly code after **;CODE**.

Note that **END-CODE** (q.v.) is not used in fig-FORTH.

;S —— 83UR, 79UR, fig. "semi S" I

Included in a mass storage block to finish interpretation of it.

< *n1, n2* —— *flag* All. "less than"

The flag is *true* if *n1* < *n2* (*false* if *n1* = *n2*).

For unsigned numbers, use **U<**.

<# —— All. "less sharp"

Used to initiate formatted output.

<BUILDS 83UR, 79UR, fig
In fig-FORTH **CREATE** must be replaced by **<BUILDS** in the **CREATE...DOES>** construction. See **DOES>**.

<CMOVE *addr1, addr2, u* —— 83 UR, 79UR, poly. "reverse C move"
The same as **CMOVE>**. See **CMOVE**.

<MARK —— *addr* 83S. "backward mark"
See **BRANCH**.

<RESOLVE *addr* —— 83S. "backward resolve"
See **BRANCH**.

= *w1, w2* —— *flag* All.
Yields *true* if *w1* = *w2*.

> *n1, n2* —— *flag* All. "greater than"
Yields *true* if *n1* is greater than *n2* (*false* if they are equal).
n1 and *n2* are signed — see also **U<**.

>BINARY *ud1, addr1* —— *ud2, addr2* poly. "to binary"
The same as **CONVERT**.

>BODY compilation address —— PFA 83R. "to body"
To do this in fig-FORTH, use **2+** .

>IN —— *addr* 83R, 79R, poly. "to in" U
A user variable whose value is the offset within the input stream showing where the next character will be read from; see Terminal Input and **BLK**. In fig-FORTH, **IN** is the word used.

>MARK —— *addr* 83S. "forward mark"
See **BRANCH**.

>R *w* —— All. "to R"
Transfers *w* to the return stack.

>RESOLVE *addr* —— 83S. "forward resolve"
See **BRANCH**.

>TYPE *addr, u* —— poly.
The same as **TYPE**, except that the string is copied to the **PAD** before being displayed.

? *addr* —— fig.
Equivalent to @ . .

?BRANCH *flag* —— 83S. "question branch"
See **BRANCH**. **?BRANCH** is a conditional version, the jump being performed only if the flag is *false* (zero).

?DUP *w* —— *w, w* (if $w \neq 0$) 83R, 79R, poly.
or 0 —— 0 "question dupe"
The top of the stack is duplicated provided that it is non-zero. This is often useful before **IF** or **UNTIL**, making **ELSE DROP** unnecessary. For example:

```
: STARS ( n ——  ) ( Prints n stars, none if n=0)
 ?DUP  IF
   0  DO  ." *"  LOOP
 THEN  ;
```

fig-FORTH uses **—DUP** equivalently.

?STACK —— *flag* poly.
Yields true if the stack is underflowed. In fig-FORTH, **?STACK** causes an **ABORT** if the stack is underflowed.

@ *addr* —— (*addr*) All. "fetch"
Used especially with variables (q.v.).

ABORT 83R, 79R, fig.
Clears the data stack and does **QUIT**.

ABORT" (run-time) *flag* ——
83R, 79UR, poly "abort quote" IC
Used within a colon definition in the form

ABORT" message**"**

At run time, if the flag is true (non-zero), the message is displayed and **ABORT** is done. For example:

```
: ?1-  ( Like 1-, but aborts if top of stack=0)
  DUP  0=  ABORT" Already zero."  1-
;
```

ABS *n* —— |n| All.
The absolute value function, which makes the top of the stack positive or zero, for example **−10 ABS** gives 10.

AGAIN (run-time) —— 83UR, 79CR, fig. IC
Equivalent to **0 UNTIL**.

ALLOT *w* —— All.
Encloses *w* bytes in the dictionary, without setting them to any special value. Note that *w* can be negative, taking bytes back out of the dictionary.

AND *w1, w2* —— *w1* AND *w2* All.
Bitwise Boolean AND. For flags, or for single bits within bit patterns, the result is *true* provided that both the arguments were.

ASCII —— *char* 83UR, 79UR. I
Used in the form **ASCII** character
The action of **ASCII** is to stack the ASCII code of the character. In compile mode, code is compiled so that this action is done not immediately but when the colon definition is executed.

ASSEMBLER —— 83A, 79A, fig, poly.
The assembler vocabulary word. It is not usually necessary to use this, since **CODE** etc. automatically switch between the assembler and FORTH vocabularies.
Note: in FORTH-79 and fig-FORTH, **ASSEMBLER** is immediate.

BASE —— *addr* All. U
A user variable containing the system number base.

BEGIN (run-time) —— All. IC

This is used to compile a repetition structure into a colon definition, using two principal forms: **BEGIN**...**UNTIL** and **BEGIN**...**WHILE**...**REPEAT**.

BEGIN...**UNTIL** repeats the stretch of FORTH program in between *until* some condition holds. The condition is left on the stack as a flag to be taken off by **UNTIL**. For example:

```
: WAIT  ( wait until RETURN is pressed)
  BEGIN
  KEY  13  =
  UNTIL  ( if key wasn't RETURN, go back to BEGIN)
;
```

Note that the repeated stretch of program is always performed at least once. See also **AGAIN** and **END**.

BEGIN...**WHILE**...**REPEAT** repeats the stretch between **WHILE** and **REPEAT** as long as the flag found on the stack by **WHILE** is true (non-zero). For example:

```
: DIGIT  ( read a digit from the keyboard)
  BEGIN
    KEY
    DUP  48  ( ASCII 0)  <
    DUP  57  ( ASCII 9)  >
    OR  ( get true if ASCII code too small or)
        ( too big for a digit)
  WHILE  ( if ASCII code not a digit then drop)
         ( it and loop back to BEGIN)
  DROP
REPEAT
  ( ASCII code for digit still on stack)
48  -  ( to get between 0 and 9)
;
```

The section between **WHILE** and **REPEAT** need never be performed at all, because if **WHILE** takes a false flag off the stack it will jump round to after **REPEAT**.

Note that the criteria for repeating are different: **UNTIL** repeats on a false flag, **WHILE**...**REPEAT** on a true flag.

BL —— 32 83CR, 79CR, fig.

A constant, the ASCII code for a space.

BLANK *addr, u* —— 83CR, poly.
Stores ASCII spaces in *u* bytes of memory starting at *addr*.
See **BLANKS**.

BLANKS *addr, n* —— 79CR, fig.
Like **BLANK**, but possibly no action if *n* is negative.

BLK —— *addr* All. "B L K" U
This user variable shows where the input stream is currently being taken from: either the text input buffer (in which case its value is 0) or a mass storage block buffer (in which case its value is the block number). In either case, **>IN** shows the offset within the buffer in use of the next character to be read.

BLOCK *u* —— *addr* All.
Leaves the address of a buffer containing the data in mass storage block u. If that block is already in a buffer, then the same buffer is used and nothing is read in from disk. (Remember that if the data has been **UPDATE**d it will no longer be the same as that on disk.)

If the block is not already in a buffer, then one is allocated to it and the data is read in. A spare buffer is used if there is one; otherwise, the one that was least recently used is selected and the block already occupying it is — if its data has been **UPDATE**d — written back to disk.

The data stays in the buffer and may be freely read and modified (see **UPDATE**). When the buffer is required for another block then, if the data has been **UPDATE**d, it is written back to disk.

The standards specify that only the last buffer address provided by **BLOCK** (or **BUFFER**) can be guaranteed to be still accurate: any use of **BLOCK** or **BUFFER** may upset all previous buffer allocations (although in practice the most recent allocations will be unaffected).

BRANCH —— 83S. C
This causes a jump (like a BASIC GOTO statement). It can be used only when compiled into a colon definition, and is there followed by a literal operand (the branch address) showing where to jump to. It is not used directly, but in the form

COMPILE BRANCH

?BRANCH is similar, doing a jump conditional on a flag from the stack.

The standard does not specify the form of the branch address, but provides four words **<MARK** and **<RESOLVE** (for a backward jump) and **>MARK** and **>RESOLVE** (for a forward jump) that cooperate in calculating it and filling it in. The MARK word always comes first, and stacks an address that the RESOLVE word can use in completing the structure.

To compile a forward jump,

```
At the jump:          COMPILE BRANCH >MARK
At the destination:   >RESOLVE
```

To compile a backward jump,

```
At the destination:   <MARK
At the jump:          COMPILE BRANCH <RESOLVE
```

They must nest properly with any other structures.

As an example, consider the **IF**...**ELSE**...**THEN** construction, which could be defined using these words if it didn't already exist. At **IF** there is a conditional forward jump to **ELSE** (or **THEN**), and at **ELSE** an unconditional forward jump to **THEN**.

```
: IF ( -- mark from forward jump)
  COMPILE ?BRANCH >MARK
; IMMEDIATE

: ELSE ( mark from IF -- mark from jump )
       ( from ELSE)
  COMPILE BRANCH >MARK
  SWAP >RESOLVE ( resolve jump from IF)
; IMMEDIATE

: THEN ( mark from IF or ELSE -- )
  >RESOLVE
; IMMEDIATE
```

Note how it is essential that these compiling words should be used in the right order. In practice, this is enforced by making the words put extra syntax checking numbers on the stack: if **IF** leaves an extra 99 (say), **ELSE** and **THEN** can **ABORT** if the 99 isn't there.

BUFFER u —— *addr* All.
This is identical to **BLOCK** except that the data is not read from disk: thus **BUFFER** is used when the block does not yet exist on disk. Make sure that if it *does* already exist, you won't mind when it gets overwritten.

C (standing for *character*) is often used as a prefix for single-byte versions of words, for example **C@**, **C!** .

C! *w, addr* —— All. "C store"
The 8 least significant bits of *w* are stored in the single byte at *addr*.

C, *w* —— 83CR, 79UR, fig, poly
Encloses a single byte in the dictionary, the 8 least significant bits of *w*.

C@ *addr* —— u All. "C fetch"
u is the single byte at *addr*, with 0s as the remaining 8 most significant bits.

CMOVE *addr1, addr2,* u —— All.
Copies one block of memory, starting at *addr1* and u bytes long, to another, starting at *addr2* (and again u bytes long). The bytes are copied starting with that at *addr1* and moving up.

Note that if the blocks overlap and *addr2* is greater than *addr1*, then the bytes towards the end of the original block will have been changed by the time they are copied. This is sometimes a useful effect, but more likely you'd avoid it by using **CMOVE>**.

CMOVE> *addr1, addr2,* u —— 83R. "C move up"
Like **CMOVE**, except that the bytes are copied in reverse order, starting at *addr1*+u−1 and moving down.

See also **>CMOVE**.

CODE 83A, 79A, fig
A defining word, used in the form

CODE name assembly code **END-CODE**

It is used to define a word in machine code instead of with a colon definition. **CODE** switches the context to the assembler vocabulary, which contains the assembly code mnemonics.

COMPILE —— All. C
This is used in a colon definition, typically that of a compiling word. It causes the following compiled address to be enclosed in the dictionary instead of being executed. For examples, see Compiling words and **BRANCH**.

CONSTANT *w* —— All.
A defining word, used in the form

w **CONSTANT** name

For example:

1024 CONSTANT 1K

The word 'name' is then a constant, with value *w*, and its action is to leave *w* on the stack:

—— *w*

Thus in the example, the constant **1K** can now be used instead of the number 1024, for example:

1K . (displays 1024)

Unlike a variable, a constant does not need @ and so it is to be preferred for values that don't change in the course of the program.

CONTEXT —— *addr* 83S, 79R, fig, poly. U
The value of this user variable specifies in some way the search context. Although details vary, fig-FORTH is typical: the value is the address of a cell in the parameter field of a vocabulary word, that cell containing the name field address of the newest word in the vocabulary.

CONVERT *ud1, addr1* —— *ud2, addr2* 83R, 79R, poly
This is used in converting a string in memory into a double length number on the stack, which may have already been partly read.
ud1 is the number converted so far.
addr1 + 1 is the address of the string.

The string is read up to the first character that is not a digit (using the system number base).

ud2 is the final accumulated number.

addr2 is the address where the reading stopped for a non-digit.

For example, suppose the five memory locations 30242 to 30246 contain the characters ".971+", and **BASE** has the value 10. If the stack has the double number 56 (presumably read from digits 5 and 6 in addresses 30240 and 30241) and (on top) the address 30242, **CONVERT** will leave the double length number 56971 and the address 30246. See **>BINARY**.

COPY *u1, u2* —— poly.
Copies mass storage block *u1* to block *u2*.

COUNT *addr* —— *addr+1, n* All.
Given a stack reference to a string, converts it from the counted form to the text form.

CR —— All.
'Carriage return'. Causes text output at the terminal to move to a new line.

CREATE — — All (but fig-FORTH is different)
A defining word, used in the form

CREATE name

Except in fig-FORTH, when executed the word 'name' stacks its PFA:

—— PFA

However, **CREATE** does not enclose a parameter field for 'name'; you are expected to do this yourself with , , **C,** and **ALLOT**. For example:

CREATE LIST 200 ALLOT

This makes space for a list of 100 2-byte numbers. To get the address of the *n*th (*n* on the stack) for use with @ or !, use

DUP + (double n) **LIST +**

See also **DOES>** for the very useful technique of defining new defining words.

In fig-FORTH, you are expected to enclose some machine code in the parameter field; the word 'name' executes its parameter field. To get the standard **CREATE**, the best way is to use **VARIABLE**, bearing in mind that **VARIABLE ALLOT**s 2 bytes of its own accord (use **−2 ALLOT** if you don't want these).

CURRENT —— *addr* 83S, 79R, fig, poly. U
The value of this user variable specifies in some way the current vocabulary. See **CONTEXT**.

D often prefixes double length arithmetic words (cf. 2).

D+ *d1, d2* —— *d1+d2* All.
Double length +.

D− *d1, d2* —— *d1−d2* 83DN, 79DN, poly.
Double length −.

D. *sd* —— 83DN, 79DN, fig, poly.
Double length . . Displays *sd* at the terminal.

D.R *sd*, u —— 83DN, 79DN, fig, poly.
Double length **.R**, displaying *sd* right aligned in u characters.

D0= *d* —— *flag* 83DN, 79DN, poly.
Double length **0=** .

D2/ *sd* —— *sd/2* 83DN. "D two divide"
Double length **2/** .

D< *sd1, sd2* —— *flag* 83R, 79R, poly.
Double length <, a signed comparison. See also **DU<**.

D= *d1, d2* —— *flag* 83DN, 79DN, poly.
Double length =.

DABS *sd* —— |*sd*| 83DN, 79DN, fig, poly.
Double length **ABS**.

DECIMAL —— All.
Makes ten the system number base.

DEFINITIONS —— All.
Whatever the context vocabulary is, **DEFINITIONS** makes it the current vocabulary, for example:

ASSEMBLER DEFINITIONS

The assembler vocabulary becomes first (by the action of **ASSEMBLER**) the context vocabulary, and then in addition (by **DEFINITIONS**) the current vocabulary.

DEPTH —— u 83R, 79R.
u is the number of stack entries (excluding itself).

DMAX *sd1, sd2* —— max (*sd1, sd2*) 83DN, 79DN, poly.
Double length **MAX**.

DMIN *sd1, sd2* —— min (*sd1, sd2*) 83DN, 79DN, poly.
Double length **MIN**.

DMINUS *sd* —— *−sd* fig.
fig-FORTH name for **DNEGATE**.

DNEGATE *sd* —— *−sd* 83R, 79R, poly.
Double length **NEGATE**; see also **DMINUS**.

DO (run-time) *limit, initial value* —— All. IC
DO is used with **LOOP** (or one of its variants) to compile a counted loop, like FOR. . .NEXT in BASIC. For example:

```
: 0TO4  ( Displays numbers from 0 to 4)
  5  0  DO
    I  .
  LOOP
;
```

Note that the *limit*, 5, is one more than the final count displayed.

At run time, **DO** takes the initial value and limit (here 0 and 5) from the stack. The stretch between **DO** and **LOOP** is executed repeatedly, under a count which takes the initial value the first time through and is incremented by 1 each subsequent time. The final time through is the last time that the count is still properly less than the limit; that is, 4 for a limit of 5. Thus in **0TO4** the loop is executed with the count equal to 0, 1, 2, 3 and 4.

I puts the count on the stack. Unlike the case in other languages, the count is stored not in a variable but in some anonymous place, usually on the return stack with the limit.

LOOP is used when the count is incremented by 1 each time. For other steps, use **+LOOP**, which takes the step off the data stack. For example:

```
: 0TO4BY2  ( Displays 0, 2 and 4)
  5  0  DO
    I  .
  2  +LOOP
;
```

Note that the step appears at the end of the loop, not at the beginning with the limit and initial value; and in fact it could be changed each time round.

For negative steps, there is the important difference that the final loop can have the count *equal* to the limit. For example (within colon definitions),

6 0 DO. . . **2 +LOOP** executes 3 times (count = 0,2,4)
0 6 DO. . . **−2 +LOOP** executes 4 times (count = 6,4,2,0)

The question of whether the count is signed or unsigned is a contentious one, and the normal answer is to make it signed for **LOOP** and **+LOOP**, unsigned for **/LOOP** (see below). FORTH-83, however, uses a *circular* count and limit, and a *signed* step.

To see how this works, consider first the case of a positive step: say **5 +LOOP**. At the end of each loop, the computer increments the count by 5 and tests to see whether this takes it up as far as the limit (in which case it carries on after **+LOOP**) or not (when it loops back to **DO**). To explain the test precisely, you must imagine that the count has **1+** done to it 5 times. If a **1+** takes it from 32767 to −32768 (treating it as signed) or from 65535 to 0 (unsigned), it doesn't matter; but if one of these **1+**s takes the count from *limit*-1 to *limit*, then the limit has been reached and there is no more

looping. Similarly for a negative step, imagine **1**—s looking for a transition from *limit* to *limit*-1.

(More practical criteria: if the step is positive, then stop if *limit* − *count* − 1 U< *step*; if the step is negative, then stop if *count* − *limit* U< −*step*.)

Some examples:

Loop form	*Number of loops*	
	FORTH-83	*FORTH-79*
30000 0 DO...LOOP	30000	30000
60000 0 DO...LOOP	60000	1
−5536 0 DO...LOOP	60000	1
0 0 DO...LOOP	65536	1
30000 −30000 DO...LOOP	60000	60000
30000 35536 DO...LOOP	60000	60000
11 10 DO...LOOP	1	1
9 10 DO...LOOP	65535	1
10 10 DO...−1 +LOOP	1	1
−10000 10000 DO...−1 +LOOP	20001	20001

In FORTH-79, if the limit is already on the wrong side of the initial value (both signed), then the loop is only executed once (it is always executed at least once). In FORTH-83, this never happens: it takes the count right round the cycle of numbers to catch up with the limit from behind.

DO loops may be nested. **I** always refers to the count of the innermost loop containing it. **J** and, in some FORTHs, **K** refer to the counts of successive outer loops.

See also **I'**, **LEAVE** and Return Stack.

DOES> All (but fig is different). IC

This defines new defining words, used in conjunction with **CREATE** in the form

: name ... **CREATE** ... **DOES>** ... ;

The part up to **DOES>** is the *defining* part: when 'name' is used to define another word, in the form

name namex

the defining part sets up for 'namex' its header (through **CREATE**) and its parameter field (through the rest).

When 'namex' is executed, its PFA is left on the stack and the part of 'name' from **DOES>** to **;** is executed. For example:

```
: ARRAY ( u ---- )( Creates an array of u 2-byte )
                  ( numbers. )
  CREATE  ( Sets up header)
  DUP  +  ALLOT  ( 2u bytes for parameter field)
DOES>  ( u, PFA ---- address of uth number)
  SWAP  DUP  +  ( doubles u )  +
;
5  ARRAY  X1  ( Numbers in X1 indexed by 0,1,2,3,4)
99  3  X1  !
3  X1  @  .  ( Displays 99)
```

This is a very simple definition of **ARRAY**: there is scope for initializing elements to 0, checking subscripts, and many other facilities.

In fig-FORTH, you must use **<BUILDS** here instead of **CREATE**.

DPL ---- *addr* 83UR, 79UR, fig. U

When **NUMBER** is used in fig-FORTH, and a decimal point is encountered in the input stream, the value of **DPL** is the number of digits to the right of the decimal point (otherwise it is −1).

DROP *w* ---- All.

Throws away the top of the stack.

DU< *ud1, ud2* ---- *flag* 83DN, 79DN, poly.

Double length U<.

DUMP *addr, n* ---- 83CR, 79CR, fig, poly.

Displays the contents of *n* bytes of memory, starting at *addr*.

DUP *w* ---- *w, w* All. "dupe"

Duplicates the top of the stack.

EDITOR —— 83CR, 79CR, fig, poly.
The editor vocabulary word.

ELSE (run-time) —— All. IC
See **IF**.

EMIT *char* —— All.
Displays the character at the terminal.

EMPTY —— poly.
FORGETs all the user's dictionary.

EMPTY-BUFFERS —— 83R, 79CR, fig, poly.
Marks all the mass storage buffers as unoccupied. No data is written back to disk.

END (run-time) *flag* —— 83CR, 79CR, fig. IC
Identical to **UNTIL**. See **BEGIN**.

END-CODE —— 83A, 79A
Encloses machine code to finish a machine code routine and return control to the FORTH system. In fig-FORTH without **END-CODE**, this machine code must be assembled in explicitly. See **CODE**.

ENDIF (run-time) —— fig. IC
Identical to **THEN**. See **IF**.

ERASE *addr, u* —— 83CR, 79CR, fig, poly.
Sets u bytes of memory to 0, starting at *addr*.

EXECUTE *addr* —— All.
Executes the word with the given compilation address. In polyFORTH, the address is the PFA.

EXIT —— 83R, 79R, poly. C
Causes premature return from the colon definition in which it occurs. It should not be used inside a **DO** loop nor when the return stack contains entries put there by **>R**.

EXPECT *addr, u* —— All.
Characters are read from the keyboard and displayed and stored in memory starting at *addr* until either u characters have been received or ENTER (a carriage return) is typed. Control characters may have special editing functions rather than being stored in memory; the carriage return is not stored (but is displayed as a space).

In FORTH-83, the number of characters stored is assigned to the user variable **SPAN**. In other FORTHs, the end of the text stored is marked with one or two nulls (ASCII 0s).

FILL *addr, u, w* —— All.
u bytes of memory, starting at *addr*, are given the value consisting of the 8 least significant bits of *w*.

FIND 83R, 79R.
Both standards use this to find the address of a word, but in rather different ways:

addr1 —— *addr2, n* 83R.

addr1 refers to a counted string, which **FIND** treats as a word and looks up in the dictionary. If the word is not defined, then *addr2* = *addr1* and *n* = 0. If the word is defined, then *addr2* is its compilation address, and *n* = 1 if the word is immediate, −1 if not. For example:

32 WORD FIND DROP

gets the compilation address of the next word in the input buffer:

—— *addr* 79R.

Used in the form **FIND** word.

addr is the compilation address of the word, or 0 if it is not defined.

FLUSH —— 83R, 79UR, fig, poly.
Writes the data from all **UPDATE**d mass storage buffers back to disk. In FORTH-83 and a number of FORTH-79 implementations, all buffers are then marked as empty. This distinguishes **FLUSH** from **SAVE-BUFFERS** and makes **FLUSH** the word to use before changing disks.

In some implementations, **FLUSH** is identical to **SAVE-BUFFERS** and so needs to be followed by **EMPTY-BUFFERS** to have the effect described here.

FORGET —— All.
Used in the form

FORGET word

This deletes from the dictionary not only the named word (its most recent definition), but also *all words defined after it*, regardless of their vocabulary. **FORGET** is a dangerous word and most systems have checks to make sure that you don't use it too cavalierly. There is usually no way of deleting individual words from the dictionary.

FORTH —— All.
The **FORTH** vocabulary word.

FORTH-83 —— 83R.
Does nothing. In an implementation not conforming with the FORTH-83 standard, the word either would not be present (so that its use would cause an error) or would cause system adjustments to ensure conformity.

H —— *addr* poly. U
The user variable whose value is given by **HERE.**

H. *n* —— 83UR, 79UR
The same as ., but in hexadecimal (base 16). The system number base is left unchanged.

HERE —— *addr* All.
addr is the address of the memory location next to be enclosed in the dictionary.

HEX —— 83CR, 79CR, fig, poly.
Makes 16 the system number base (hexadecimal notation).

HOLD char —— All.
Used in Formatted Output.

I —— loop count All.
See **DO**. Note that **I** is usually equivalent to **R@**.

I' —— loop limit 83UR, 79 UR, poly.
Stacks the limit of the innermost **DO** loop containing it. This is usually the second from top on the return stack.

IF (run-time) *flag* —— All. IC
This is used in colon definitions with **THEN** and (optionally) **ELSE** to provide conditional execution. For example:

```
: NEGPOS ( n ―― ) ( Announces the sign of n)
  0< IF ( if n < 0)
    ." negative "
  ELSE ( if n ≥ 0)
    ." positive or zero "
  THEN
;
```

If (at run-time) **IF** finds on the stack a *true* flag (or any non-zero number) it performs the part between **IF** and **ELSE** and skips over to **THEN**, while if it finds a *false* flag (zero) it skips over to **ELSE** and carries on. In either case, control eventually reaches **THEN** and then carries on as normal.

There is an alternative form that omits **ELSE**, for example:

```
: NNEG? ( n ―― ) ( aborts if n<0)
   0< IF ABORT THEN
;
```

Here, if **IF** finds a *false* flag, it skips over to **THEN**.

Note the unusual order in FORTH compared with other languages:

condition **IF** true part **ELSE** false part **THEN**
or condition **IF** true part **THEN**

IMMEDIATE —— All.
The most recently defined word is made immediate. For examples, see **BRANCH**, **[COMPILE]** and Compiling words.

INDEX *u1, u2* —— 83UR, 79UR, fig.
Displays the first lines of consecutive mass storage blocks starting with block *u1* and finishing with block *u2*. It is common practice to put a title or comment on the first line of each block.

INTERPRET —— 83CR, 79UR, fig, poly.
Applies the text interpreter to the remainder of the input stream specified by **BLK** and **>IN**.

J —— *loop count* All. C
Stacks the count of the second innermost **DO** loop containing it. See **DO**. This is often third from top on the return stack.

K —— *loop count* 83CR, 79UR. C
Stacks the count of the third innermost **DO** loop containing it. This is often fifth from top on the return stack.

KEY —— *char* All.
Waits for a key to be pressed at the terminal and stacks its ASCII code. The 9 most significant bits may vary from system to system, so to get the true ASCII code use

KEY 127 AND

There is no standard way of reading the keyboard instantaneously (for example, for a game). Most systems provide words to do this; typical ones are **INKEY** and **?KEY**.

LEAVE (run-time) —— All. C (sometimes I)
(compile-time) ——
Causes a premature termination of the innermost **DO** loop containing it. In FORTH-83, **LEAVE** causes a jump to just beyond the appropriate **LOOP** or **+LOOP** and is a compiling word. It does not use the data stack at compile time, and this allows it to be used within other program structures nested in the **DO**...**LOOP**.

In older FORTHs, **LEAVE** makes the limit equal to the count so that **LOOP** or **+LOOP** is bound to stop looping, but execution carries on up to the **LOOP** or **+LOOP** in the usual way.

For example, this word **TOTAL** totals consecutive numbers from a stretch of memory, stopping early if a 0 is found.

```
: TOTAL  ( final address+2, initial address  ——  )
  0  ( initialize total)
  ROT  ROT  DO
    I  @  ( next number)
    DUP  0=  IF  LEAVE  THEN
    ( If in FORTH-83, and have left early because)
    ( of a 0, + won't be executed.                )
    +
  2  +LOOP
;
```

See Return Stack.

LIST u —— 83CR, 79R, fig, poly.
Displays (as ASCII text) mass storage block u, and assigns u to **SCR**.

LITERAL (compile-time) *w* —— All. IC
(run-time) —— *w*
Compiles a number from the stack, as though it had been typed in in compile mode. For example:

```
90  CONSTANT  RIGHT-ANGLE
: REVS  ( n  ——  n*4*RIGHT-ANGLE)
    [  4  RIGHT-ANGLE  *  ]  LITERAL  *
    ( equivalent to 360 *)
;
```

Note that at compile time : and compiling words use the stack in unspecified ways, so it is best to put *w* on the stack immediately before **LITERAL**. For instance, this next definition is unlikely to work:

```
4  RIGHT-ANGLE  *
: REVS  LITERAL  *  ;
```

LOAD u —— All.
Applies the text interpreter to mass storage block u, thus loading it in as FORTH text.

LOOP (run-time) —— All. IC
Equivalent to **1 +LOOP**. See **DO**.

M* *n1, n2 —— n1*n2* fig, poly.
The product is signed, double length (cf. **UM***).

M*/ *sd, n1, n2 —— d*n1/n2* poly.
Similar to */, but *sd* and the result are signed double length, and the intermediate product *sd*n1* is triple length.

M+ *d, n —— d+n* poly.
The result is double length.

M/ *sd, n —— sd/n* poly.
The quotient is signed single length.

sd, n —— r, q fig.
r and q are the remainder and quotient, both signed single length, for the division *sd/n*.

MAX *n1, n2 —— max (n1, n2)* All.
The larger of *n1* and *n2* (signed).

MIN *n1, n2 —— min (n1, n2)* All.
The smaller of *n1* and *n2* (signed).

MINUS *n —— −n* fig.
Identical to **NEGATE**.

MOD *n1, n2 —— n1* MOD *n2* All.
The remainder when *n1* is divided by *n2*. See **/MOD**, which explains the sign conventions used.

MOVE *addr1, addr2, u ——* 83UR, 79R, fig, poly.
This is a cell (2-byte) version of **CMOVE**, but the significance of u varies. In FORTH-79 and fig-FORTH, u is signed and gives the number of cells to be copied (none if u is not positive), while in the FORTH-83 reference set and polyFORTH it is the number of bytes (but copying is done a cell at a time in distinction from **CMOVE**).

NEGATE *n —— −n* 83R, 79R, poly.
See also **MINUS**.

NOT All.
This is different in FORTH-83 from previous FORTHs:

w —— NOT *w* 83R

Bitwise Boolean NOT, or one's complement. It complements (changes 0 to 1 or 1 to 0) the 16 bits of *w*.

w —— *flag* 79R, poly.

In older FORTHs, **NOT** is *not* bitwise Boolean, but is identical to **0=**.

To get the bitwise Boolean effect, use one of the following three combinations:

−1 XOR
NEGATE 1−
MINUS 1−

NUMBER *addr* —— *sd* 83UR, 79UR, fig, poly.
Converts text starting at *addr* (possibly with a minus sign) to a signed double length number *sd*.

In fig-FORTH, a decimal point '.' in the number is ignored, except that the number of digits after it is assigned to **DPL**.

In polyFORTH, the number is double length only if it contains ':', ',', '−' (except initially), '.' or '/', the number of digits after which is assigned to **PTR**. Otherwise the number is left single length, with **PTR** negative.

All conversion is according to the system Number Base.

O. *n* —— 83UR, 79UR
Like . , but always in octal (base 8) without changing the system Number Base.

OCTAL —— 83CR, 79UR, poly.
Makes 8 the system number base.

OFFSET —— *addr* 83CR, 79CR, fig, poly. U
A user variable whose value is added to mass storage block numbers to obtain block numbers on disk. For instance, this can be changed to switch between two disk drives.

OR *w1, w2* —— *w1* OR *w2* All.
Bitwise Boolean OR. For flags, or for single bits within bit patterns, the result is true provided that either of the arguments is (or both are).

OVER *w1, w2* —— *w1, w2, w1* All.

PAD —— *addr* All.
The address of a workspace area (usually at least 64 bytes). This is typically a fixed distance beyond the end of the dictionary, so its address or contents may change if the dictionary is extended or if you use **FORGET**. The system is also likely to use the pad for numeric output, formatted or unformatted.

PAGE —— 83UR, 79UR, poly.
Clears the terminal screen.

PICK *n* —— *w* 83R, 79R
Note that the two standards define **PICK** differently.
w is the *n*th entry down from the top of the stack, not counting *n* itself. The top of the stack is numbered 0 in FORTH-83, 1 in FORTH-79. For example:

Stack before **4 PICK** : 6 5 4 3 2 1
Stack after **4 PICK** :
(FORTH-83) 6 5 4 3 2 1 5
(FORTH-79) 6 5 4 3 2 1 4
Some equivalences:

	FORTH-83	*FORTH-79*
0 PICK	**DUP**	error
1 PICK	**OVER**	**DUP**
2 PICK		**OVER**

fig-FORTH and polyFORTH often haven't got **PICK** and it is then usually easiest to make greater use of **R>** and **>R** instead.

QUERY —— 83CR, 79R, fig.
Erases the text input buffer, then uses **EXPECT** to put in it up to 80 characters from the keyboard.

QUIT —— All.
Clears the return stack and text input buffer and returns control to the terminal in execute mode.

R# —— *addr* poly. U
User variable whose value is the current character position in the editor.

R> —— *w* All. "R to"
Takes off the top of the return stack and transfers it to the data stack.

R@ —— *w* 83R, 79R. "R fetch"
Copies the top of the return stack onto the data stack. In fig-FORTH and polyFORTH, **I** has the same effect.

RECURSE 83CR. IC
This is used within a colon definition instead of the word itself (which is smudged and thus unrecognizable) to implement simple recursion. **SELF** is often used with the same meaning.

REPEAT (run-time) —— All. IC
See **BEGIN**.

ROLL u —— 83R, 79R
Note that the two standards define **ROLL** differently.

ROLL is like **PICK** (q.v.), except that the *u*th entry, brought to the top, is removed from its place lower down in the stack. For example:

Stack before **4 ROLL** : 6 5 4 3 2 1
Stack after **4 ROLL** :
(FORTH-83) 6 4 3 2 1 5
(FORTH-79) 6 5 3 2 1 4

Some equivalences:

	FORTH-83	*FORTH-79*
0 ROLL	no change	error
1 ROLL	**SWAP**	no change
2 ROLL	**ROT**	**SWAP**
3 ROLL		**ROT**

ROT *w1, w2, w3* —— *w2, w3, w1* All. "rote"
The third stack entry down is moved up to the top.

S0 —— *addr* 83UR, 79UR, fig, poly. U
The value of this user variable is the address of the bottom of the stack.

SAVE-BUFFERS —— 83R, 79R
Normally this is like **FLUSH** except that afterwards the buffers are not deallocated (but check this on your implementation). **SAVE-BUFFERS** should be used periodically during an editing session to ensure that the disks are kept up to date (as a precaution against system crashes).

SCR —— *addr* 83UR, 79UR, fig, poly. U
The value of this user variable is the number of the mass storage block most recently **LIST**ed. (Editor commands will usually apply to this block.)

SIGN
Inserts the sign in formatted output; fig-FORTH and polyFORTH are non-standard.

n —— 83R, 79R

If n is negative, **HOLD** is a minus sign. For example, to print a signed double length number divided by 100, with two decimal places,

```
: 100/.  ( sd     --     )
   DUP  >R  ( store more significant half on return)
            ( stack to show sign)
   DABS
   DECIMAL  <#
     #  #  ( two decimal places)
     46  HOLD  ( decimal point)
     #S  ( remaining digits)
     R>  SIGN
   #>  TYPE
;
```

n, ud —— *ud* fig, poly.

This is the same as in the standards, except that the number *n* whose sign is used is expected to be *third* on the stack.

SP@ —— *addr* 83UR, 79UR, fig. "S P fetch"
The address of the top of the stack, before **SP@** was executed.

SPACE —— All.
Displays a single space.

SPACES *u* —— All.
Displays *u* spaces.

SPAN —— *addr* 83R. U
See **EXPECT**.

STATE —— *addr* 83R, 79R, fig. U
The value of this user variable is 0 during execute mode, non-zero during compile mode.

SWAP *w1. w2* —— *w2. w1* All.
Swaps the top two stack entries.

TEXT *char* —— 83UR, 79UR, poly.
Like **WORD**, except that

- The text is copied to the pad. The address is not left on the stack, but can be obtained from **PAD**.
- The rest of the pad is cleared out to spaces.

THEN (run-time) —— All. IC
See **IF**. Some FORTHs use **ENDIF** equivalently.

THRU *u1, u2* —— 83CR, 79UR
LOADs consecutive mass storage blocks, starting with block *u1* and finishing with block *u2*.

TIB —— *addr* 83R, fig.
In FORTH-83, *addr* is the address of the start of the *text input buffer*, where keyboard input is stored for the input stream. In fig-FORTH, **TIB** is a user variable whose value is this address (so where in FORTH-83 you'd use **TIB**, in fig-FORTH you'd use **TIB** @). For example (in FORTH-83, and assuming **BLK** has the value 0),

```
TIB >IN @ + C@
```

gives the ASCII code of the next character in the text input buffer.
In strict FORTH-79, you cannot access the text input buffer.

TYPE *addr, u* —— All.
The arguments refer to a text string. **TYPE** displays this string (that is, it displays *u* characters starting at *addr*).

U* *u1, u2* —— *u1***u2* 79R, fig, poly.
Identical to **UM***.

U. *u* —— 83R, 79R, poly.
Unsigned version of . . Displays *u* followed by a space.

U.R *u1, u2* —— 83CR, 79CR, poly.
Unsigned version of **.R**. Displays *u1* right aligned in a field of *u2* characters.

U/ ud, u —— r, q fig.
Identical to **UM/MOD**.

U/MOD ud, u —— r, q 79R, poly.
Identical to **UM/MOD**.

U< u1, u2 —— *flag* All.
Unsigned version of <: thus **1** **−1** **U<** returns a *true* flag, because −1 is treated as the unsigned number 65535.

UM* u1, u2 —— u1*u2 83R
The result is the unsigned, double length product. See also **U***.

UM/MOD ud, u —— r, q 83R
Divides ud by u to give a quotient q and remainder r. All four numbers are unsigned; ud is double length and the rest are single length. See also **U/** and **U/MOD**.

UNTIL (run-time) *flag* —— All. IC
See **BEGIN**. Some FORTHs use an equivalent word **END**.

UPDATE —— All.
Marks the currently valid mass storage block (the one most recently referenced by **BLOCK** or **BUFFER**) as **UPDATE**d: you should do this if you change the data in the buffer and wish eventually to have the changes written back to disk. The changes are written back if the buffer is allocated to another block, but it is safer to force this with **SAVE-BUFFERS** or **FLUSH**.

Editor programs will usually do **UPDATE** automatically.

USER u —— 83UR, 79UR, fig, poly.
Most implementations of FORTH keep an area of memory, separate from the dictionary, for the values of user *variables* such as **BASE**, **SPAN**, **>IN** etc., and often there is spare space there for your own uses. **USER** is a defining word to define new user variables, and it is used in the form

u **USER** name

u is the offset within the user area where the value of 'name' is to be stored. (Contrast this with an ordinary variable, whose value is stored in its parameter field in the dictionary.)

When executed, 'name' leaves the address of its value on the stack, just like an ordinary variable (see **VARIABLE**).

VARIABLE —— All (but fig is different)
A defining word to set up a variable, used in the form

VARIABLE name
e.g. **VARIABLE HORSES**

(In fig-FORTH, this is preceded by a value used to initialize the variable.)

HORSES can now be given a value (which is stored in its parameter field) using **!**, for example:

8 HORSES ! (like BASIC LET HORSES=8)

and its value can be retrieved using **@**, for example:

HORSES @ . (displays 8)

HORSES itself just leaves its parameter field address on the stack, so you must remember to use **!** or **@** to deal with its value.

See also **? +! CONSTANT 2VARIABLE 2CONSTANT USER**.

VLIST —— 79UR, fig.
Displays a list of the words in the search context. There are usually keys to press to stop the list temporarily, to resume it, and to stop it permanently. See also **WORDS**.

VOCABULARY —— All.
A defining word to define a new vocabulary, used in the form

VOCABULARY name

In fig-FORTH (and others), 'name' is chained to its *parent* (the current vocabulary in which it was defined), so that any dictionary search through the 'name' vocabulary follows up with a search through its parent.

WHILE (run-time) *flag* —— All. IC
See **BEGIN**.

WIPE —— poly.
Blanks out the current mass storage block (last referenced by **BLOCK** or **BUFFER**) in its buffer.

WORD *char* —— *addr* All (but fig is different)
Reads the next word from the input stream and copies it as a counted string to the memory locations starting at *addr*.

The role of the space as delimiter of words can be taken over by any other character as specified by *char* — for instance, ." , (and .(use **WORD** with '"' or ')' as the delimiter *char*.

More precisely, **WORD** reads from the input stream:

(1) Any initial delimiter characters *char*.
(2) Text up to but not including a delimiter or the end of the input stream.
(3) A single delimiter, if there is one, after the text.

The text in (2) is copied as a counted string followed by a space (not included in the count), for example:

46 WORD...catnip.**COUNT TYPE**

displays 'catnip' (46 is ASCII code for '.').

In fig-FORTH, **WORD** does not leave the address; as in many other implementations, the counted string is always at **HERE**.

WORDS —— 83UR
Similar to **VLIST**.

XOR *w1, w2* —— *w1* XOR *w2* 83R, 79R, fig.
Bitwise Boolean exclusive OR. For flags, or for single bits within bit patterns, the result is *true* provided that one of the arguments is *true*, but not both of them.

[—— All. I
Switches to execute mode. For example:

```
: CHEQUER ( addr —— )
  ( Sets 8 consecutive memory locations alternately)
  ( to 55 hex and AA hex.)
  DUP 8 + SWAP DO
    [ 16 BASE ! ] ( This is done at compile time)
    55 I C! AA I 1+ C!
  [ DECIMAL ] 2 +LOOP
;
```

See **LITERAL** for another example.

Note that : and compiling words often use the stack at compile time, so don't let [. . .] foul it up for them (cf. **LITERAL**).

['] (run-time) —— *addr* 83R, poly. IC
A compiling word, used in a colon definition in the form
['] word

Instructions are compiled so that at run-time the compilation address (or, in polyFORTH, the PFA) of the word is put on the stack. For example:

```
VARIABLE EXECUTION-VECTOR
: DODUP  ( makes EXECUTION-VECTOR refer to DUP)
  ['] DUP  EXECUTION-VECTOR  !
;
```

Later, **EXECUTION-VECTOR @ EXECUTE**
does **DUP** if **DODUP** has been used.

In fig-FORTH and FORTH-79, the compile mode action of ' does the same (with PFAs).

[COMPILE] —— All. IC
A compiling word, used in a colon definition in the form

[COMPILE] word

It specifies that 'word', even if immediate, is to be compiled and not executed. For example (as part of a French redefinition of FORTH):

```
: SI [COMPILE] IF ; IMMEDIATE
```

For more examples, see Compiling words.

] —— All.
Switches into compile mode. See **[**.

Pitman Pocket Guides

The complete list of titles in this series is printed on the stiff board at the back of this Guide.

This series of pocket size reference guides provides you with reliable descriptions of the salient features of all the important languages, micros, operating systems and word processors. You can use them as memory-joggers or reference tools.

There is an introductory Guide to each category for those who have little experience of the subject. This provides you with the lead-in to other related titles.

The Publishers would welcome suggestions for further improvements to this series. Please write to Alfred Waller at the address below.

PITMAN PUBLISHING LTD
128 Long Acre, London WC2E 9AN

Associated companies
Pitman Publishing Pty Ltd, Melbourne
Pitman Publishing New Zealand Ltd, Wellington
Copp Clark Pitman, Toronto

Consultant Editor: David Hatter

First edition 1984

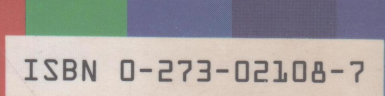
Printed in Great Britain at The Pitman Press, Bath

ISBN 0 273 02108 7

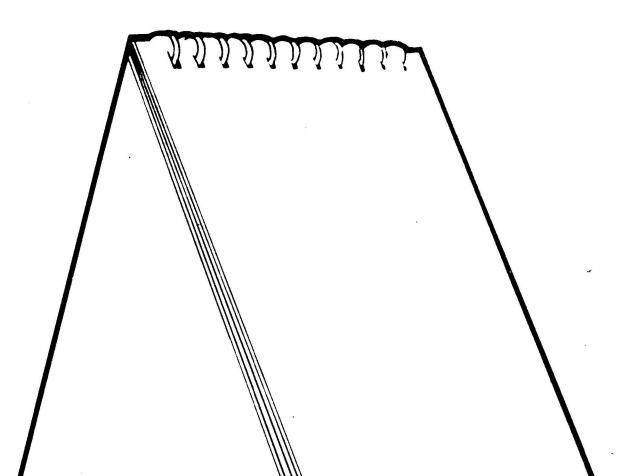
Index

Pitman

ISBN 0-273-02108-7



FORTH



The diagram shows, how to arrange the Pocket Guide in an upright position.

Bend at score mark indicated by arrow.

Pitman Pocket Guide Series

Programming	
Programming	John Shelley
BASIC	Roger Hunt
COBOL	Ray Welland
FORTRAN	Philip Ridler
Pascal	David Watt
FORTRAN 77	Clive Page
LOGO	Boris Allan
FORTH	Steven Vickers
Assembly Languages	
Assembly Language for the 6502	Bob Bright
Assembly Language for the Z80	Julian Ullmann
Assembly Language for the 8085	Noel Morris
Assembly Language for MC68000 Series	Robert Erskine
Microcomputers	
Programming for the BBC Micro	Neil Cryer and Pat Cryer
Programming for the Apple	John Gray
Sinclair Spectrum	Steven Vickers
Commodore 64	Boris Allan
Acorn Electron	Neil Cryer and Pat Cryer
The IBM PC	Peter Gosling
Operating Systems	
Introduction to Operating Systems	Lawrence Blackburn and Marcus Taylor
UNIX	Lawrence Blackburn and Marcus Taylor
CP/M	Lawrence Blackburn and Marcus Taylor
MS-DOS	Val King and Dick Waller
PC-DOS	Val King and Dick Waller
Word Processors	
Introduction to Word Processing	Maddie Labinger
WordStar	Maddie Labinger
Wang System 5	Maddie Labinger
IBM Displaywriter	Jacquelyne A Morison
Philips P5020	Peter Flewitt